Sujal Gupta: EE22BTECH11052
Aditya Vikram Singh: EE22BTECH11001

# Computer Architecture - CS2323, Autumn 2024

**Lab-7 (Cache Simulator)**

**Report**

## Cache Implementation

The cache used is a 3D vector with the 1st dimension representing the number of indexes, 2nd dimension representing the number of ways (associativity), and the 3rd dimension representing the number of bytes.

1. **Read Instructions:**
   1. Based on the command (cmd), we determine the type of load (lb, lh etc.) and the number of bytes (no_bytes) to retrieve from cache.
   2. If the cache is disabled, we directly take the data from the memory, skipping the cache.
   3. Now for the given source address we define the number of Tag, Index and the offset bits.
   4. Now it searches the cache for a hit or miss; on a cache Hit it Loads the data from cache into the register and updates the replacement policy, on a cache Miss it selects a cache line for replacement based on the configured replacement policy.
   5. Now, we load the data into the cache from memory, updating whether it is hit or miss, and loads data into the destination register from cache.
   6. We define a "getvaluefromcache" function which gets data from the cache for the given address, byte size, cache index, and way, then it Reads bytes from the specified cache line, based on the offset (addr_offset), for finding the final value.
   7. We define a "load_from_cache" function which handles load instructions (like lb, lh, lw, etc.) by calling getvaluefromcache based on the instruction type, it handles both signed and unsigned by doing sign extension.
   8. And for a miss case, we define the function "miss_case_load" which updates the cache with the data taken from memory, updates the valid bit, and stores the tag. Make the dirty bit as 0 for a new load instruction.
   9. For finding the offset, index and tag we use the following formulaes:

$$ind = {tot\_size}/{\text{associativity} * \text{block\_size}}$$

$$ind\_bits = \log_2 ind$$

$$\text{way\_bits} = \log_2 ways$$

$$\text{offset\_bits} = \log_2 off$$

   10. As an exception for fully associative where the associativity is given as 0, we make the ind as 1, and associativity as tot_size/block_size.

2. **Write Instructions:** For the write instructions, as per given in the problem statement, we have implemented write-through with no allocate condition and write-back with allocate condition, given below are the steps for our implementation:
   Similar to load instructions, we define the functions store_to_cache and store_handle which finds the offset, index, and tag components to locate the correct cache block.
   1. **Write-through:**
      a. **Write-through with cache miss:** If a cache miss occurs, we just directly update the main memory and increase the miss count.
      b. **Write-through with cache hit:** If a cache hit occurs, we update the data in the cache block and the dirty bit of the cache block to made 1, main memory is also updated immediately, and the hits are increased.

   2. **Write-Back:**
      a. **Write-back with cache miss:** When a write operation encounters a cache miss, it gets the required block from the main memory, then we find the index of the cache block using the given replacement policy (e.g., LRU_update etc). then the cache is updated, and the dirty bit of the block is made 1, indicating that the cache contains new data not yet written back to main memory. During a miss-based replacement, if the replacement policy selects a cache block with a dirty bit set to 1, write the modified data from that block back to main memory before replacing it, and the miss count is increased.
      b. **Write-back with cache hit:** If a cache hit occurs, we directly update the data in the cache block and the dirty bit of the cache block to made 1, main memory is not updated immediately, and the hit count is increased.

**Overheads:**

1. **Valid Bit:** For denoting the valid bit for an index, we use a 2D array with the rows representing the index and columns representing the number of ways. (In case of direct-mapped it acts as a 1D array as ways = 1).
2. **Dirty Bit:** For denoting the dirty bit for an index, we use a 2D array with the rows representing the index and columns representing the number of ways. (In case of direct-mapped it acts as a 1D array as ways = 1).
3. **Replacement Table:** For denoting the replacement bit for an index, we use a 2D array with the rows representing the ways columns representing the index. (In case of direct-mapped it acts as a 1D array as ways = 1), now depending upon the replacement policy the table is updated.

**Replacement Policy:** We use "repl_policy_num" in the code for identifying the replacement policy from the given config file, i.e. FIFO -> 1, LRU -> 2, RANDOM -> 3.

1. **LRU:** The LRU replacement policy is implemented by maintaining a count of how "recently" each line in the set was accessed. After each access (hit

`

or miss), the LRU values are updated such that the line with the highest count is always the least recently used and hence eligible for replacement.

1. If we encounter a miss case in read or write instructions, we give the input argument row as -1, then this function Searches for the line in the set with the highest LRU value (associativity - 1), which represents the least recently used line, it sets miss_index to this line index and resets the LRU value of this line to 0 (indicating it has just been accessed), For all other lines in the set, it increments the LRU value if it is less than val (associativity - 1) in this case (representing they have been used less recently than the accessed line).
2. Now if we encounter a hit case i.e. row != -1, then val is set as the current LRU value in the accessed line, For all other lines in the set, it increments the LRU value if it is less than val (representing they have been used less recently than the accessed line). Finally, it sets the LRU value of the accessed line to 0, marking it as the most recently used.
3. The function identifies the line with the maximum LRU value in the set (associativity - 1) as the replace_index. This line is selected for replacement in the next cache miss for this set.
4. If a miss_index was identified during a cache miss, lru_replace_index is set to miss_index. Otherwise, lru_replace_index is set to replace_index, which will be used in a future replacement decision.

2. **FIFO:** The fifo_update function manages cache line replacement using a First-In-First-Out (FIFO) policy. It first checks for an empty slot in the cache set. If an empty slot is found, it fills it with the current timestamp, updates the timestamp tracker for that particular index, and exits. If no empty slot is available, it identifies the oldest cache line (the one with the lowest timestamp), replaces it, and updates the timestamps of other lines to maintain the FIFO order. The function returns the index of the replaced or newly filled cache line, ensuring that the oldest entry is always replaced first when the set is full.
3. **RANDOM:** It just randomly allocates a number between 0 and ways – 1, using the rand() function.

**Note:** We have implemented all the three parts of the given problem statement.

**Testing:**

the code written was tested against various possible cases at each point for each kind of instruction format to detect errors. The output produced by our code was checked against the output of the RIPES cache simulator.

`